Proceedings of the 2009 IEEE International Conference on Systems, Man, and Cybernetics
San Antonio, TX, USA - October 2009

Efficient Optimized Composition of Semantic Web Services

Rattikorn Hewett, Bach Nguyen and Phongphun Kijsanayothin
Department of Computer Science, Texas Tech University
Rattikorn.Hewett@ttu.edu, Bach.Nguyen@ttu.edu, kphongph@gmail.com

***Abstract*— Advances in Internet and software technology play increasing roles in creating work environments and online services that impact our everyday living. Web service computing paradigm has revolutionized how these software applications can be developed rapidly and reliably by employing available services available in standard forms across the web. Automatic composition of web services is necessary and challenging when there are a huge growing number of available web services but no single service offers the desired function. This paper presents an approach to automatically combining published web services to meet the application requirements using as few services as possible. Our approach is based on heuristic algorithms that incorporate semantics of services from their ontology into identifying valid executable services in the composition structure. The paper describes the approach and evaluates its performance on public service repositories. Our experimental results show correct optimized solutions 100% of the time, with a reduction in the average running time, compared to champion of Web service challenge'06 competition, of 60.2% over 31 composition problems.**

Keywords: **semantic web services, automatic service composition, service-oriented computing, ontology, heuristic search.**

I. INTRODUCTION

Advances in Internet and software technology play increasing roles in creating work environments and online services that impact our everyday living. It is no longer necessary to go to bookstores to buy books or to visit banks to complete certain transactions. Many types of services can be obtained any time and anywhere that are networked accessible across the Internet. Modern enterprises rely on software to conduct business or perform day-to-day operations. Web service computing paradigm has revolutionized how developers can rapidly and reliably create these service-oriented software applications by employing available web services. The term *web services* (or *services*) refer to self-contained platform independent software units prescribed by standard machine-readable specifications and protocols to support interoperable services for distributed applications across the Web [4, 6].

When there is a huge growing number of available web services that no single service can offer the desired function, automatic composition of web services becomes necessary and challenging. Current approaches to automatic web service composition in the literature (see [1, 4, 6, 7] for surveys) often include service discovery techniques for selecting appropriate services and integrating them into a composite service structure that has been *pre-specified* [7]. In *syntactic composition,* syntactical languages such as BPEL and WS-CDL are used to specify interfaces and process flows of combined services, while *semantic composition* relies on DAMLS (ServiceModel), for specifying semantics of service processes along with service ontology languages (e.g., OWL-S and WSMO) for guiding inferences in planning systems (e.g., Golog, HTN) [3, 6]. However, most of these approaches require manually written templates of composite structures. No assembling of complex flows from atomic service message exchanges is automatically derived from a search process [7]. Other automated approaches to web service composition are based on formal methods including logical inductions and model checking for deriving composition paths [4]. However, they tend to be very complex and computationally expensive. Furthermore, service compositions based on rules, or logical planners [3, 4, 7, 8] often rely on representational models that may be suitable for planning problems but unnecessarily complex for service composition problems.

This paper addresses the issue of semantic web service composition, specifically how to efficiently and automatically construct a valid executable composition (sequence) of published services to satisfy the application requirements with minimal number of services. Our approach is based on an effective modeling approach and two heuristic algorithms that incorporate semantics of services from their ontology into identifying valid executable services in the composition structure. The rest of the paper is organized as follows. Section II describes related work including champion of Web service challenge'06 competition [10] system by Weise et al. [9], which we use in our comparison study. Section III defines the problem and introduces our modeling approach. Section IV presents our composition approach along with its complexity analysis and Section V illustrates its use for solving service composition problems. Section VI empirically evaluates the approach by comparing our results with those obtained by Weise et al.'s system. The paper concludes in Section VII.

II. RELATED WORK

Several automated approaches to semantic service composition have been proposed [3, 5-9, 11, 12]. Most use backward search control, like in planning systems, and indexing mechanisms to improve efficiency. The optimization of the number of services deployed in a composition has been studied using graph-based [9] and planning-based [5] approaches.

Weise et al. [9], champion of WS-challenge06 competition, employ three alternatives: iterative deepening depth first search, greedy search, and genetic algorithms. The greedy search usually outperforms the rest. Weise et al.'s greedy algorithm searches in a backward fashion using a heuristic that prefers a service that produces a maximum number of parameters for a current goal, which is updated (for the next move) to include the input parameters required by the selected service. Our approach, however, searches forwardly. It prefers a service that produces the highest number of (1) new desired parameters, and (2) new parameters when results from (1) are the same.

Oh et al. [5] extend WSPR, a planning-based web service composition to handle semantic service composition. WSPR uses forward and backward search and a heuristic-based regression search that prefers a web service with a large coverage set of the desired output parameters. The overall performance is dominated by the forward search step that, for a repository of n services and m parameters, takes $O(n^2m)$, a polynomial time of a higher degree than that of our approach. Hoffman et al. [2] propose a forward search approach that takes polynomial time reasoning for semantic service composition (not necessary optimal). However, the analysis assumes that discovery to be a pre-process step. Unlike [2], our polynomial time complexity does not require this assumption. Furthermore, we aim to optimize the composition.

III. Problem Formulation and Modeling Approach

A. The Optimized Semantic Service Composition Problem

Given a service repository W over a set of parameters P, where each web service $w \in W$, is described by its corresponding set of input (output) parameters $in(w)$ ($out(w)$) $\subseteq P$. Let KB be a knowledge base containing an ontology in a web service domain. In a context of web service composition, the ontology typically represents models of type and part-whole hierarchies of service parameters and relevant concepts at various abstraction levels. Thus, KB, over a set of parameters P, describes the semantic of parameters in P.

Suppose we want to develop a composite web service C with a set of input (output) parameter requirements I (O) $\subseteq P$. Given a service repository W and an ontology knowledge base KB, over a set of parameters P, an *optimized semantic web service composition problem* for a composite service C is to find an executable sequence of web services from W that takes I to produce O with a minimal number of services. Such an executable sequence is obtained by exploiting information from the web service ontology in KB.

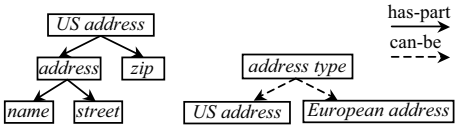


Figure 1. Example of an ontology *KB*.

To see the impact of the ontology, consider a trivial service composition problem for a composite service C specified by $I = \{address\ type\}$ and $O = \{name, street, zip\}$ for a given $W = \{w \mid in(w) = \{USaddress\}, out(w) = \{address, zip\}\}$ and a KB as shown in Figure 1. The ontology allows us to infer that *address type* can be a *US address* making w applicable for I of C. Furthermore, we can also infer that *address* consists of *name* and *street* and therefore, O of C can be satisfied. Without the ontology, the development of service C would have been viewed as unsuccessful.

B. Service Composition Modeling Approach

We propose a state space search model where a state represents a set of callable parameters of the composite service application developed so far along with their related concepts obtained by service ontology. Each transition from one state to another represents an applicable web service to be deployed in the service composition. In other words, the model consists of a set of *states* S with an *initial state* $s_0 \in S$, a set of *goal states,* $G \subseteq S$, and a set of *actions* (or *moves*) W representing web services available in a repository. We define $p(s) \subseteq P$ to be a set of *callable parameters* in state s. Thus, $p(s_0) = I$ and $G = \{s \in S \mid O \subseteq p(s)\}$.

To exploit service semantics, for a given set of concepts A in a service ontology KB, we define *infer-semantics*(A) to be a set of inferred concepts of A. For example, for a KB in Figure 1, *infer-semantics*({*US address, European address*}) produces {*address*, *zip*, *name*, *street*}. Thus, *infer-semantics* includes all successors of a given set of concepts, some of which are service parameters. As in a typical interpretation, a part-whole (type) hierarchy exhibits a logical AND (OR) of all inferred concepts.

A web service w is *applicable* to state s, if $in(w) \subseteq p(s)$. This is called *syntactic matching.* Similarly, for *semantic matching*, we replace $p(s)$ by $p(s) \cup$ *infer-semantics*($p(s)$). A *transition* δ: $S \times W \rightarrow S$ moves one state to another by applying an applicable appropriate service action. Note that if $\delta(s, w) = t$ then $p(t) = p(s) \cup out(w)$. In general, $p(s)$ collects all callable parameters along all possible paths from s_0 to s. Thus, $p(s)$ exhibits a monotone (non-decreasing) property. This property also holds for semantic matching (i.e., $p(s) \cup$ *infer-semantics*($p(s)$)) and in dynamic composition (where determining applicable services are checked from dynamically updated service repository) when published web service changes do not retract old parameters. In Section *C* of Part IV, we will show that the monotone property is crucial to the efficiency of our proposed approach.

IV. Automatic Semantic Service Composition

Our approach to automatically constructing a valid executable web service composition structure consists of two basic steps: (1) finding an executable sequence of web service applications that satisfy a given composite service requirements, and (2) pruning unnecessary web services in the sequence obtained in Step (1) so that it has (near) minimal number of services. Step (1) is based on *Build-sequence*, a greedy search algorithm, as described in Section *A*. For Step (2), to obtain a (near) minimal sequence of service composition, we propose

the *Prune-sequence* algorithm as described in Section *B*. Section *C* gives complexity analysis of the proposed algorithms.

A. Constructing Service Composition

Basic steps of the *Build-sequence* algorithm are shown in Figure 2. In Line 2, the search starts from an initial state that represents an initial set of callable parameters from a given set of required input parameters I of a composite service to be developed along with corresponding elaborated concepts derived from the service ontology. To advance to each next state, *build-sequence* employs heuristic to select an appropriate applicable service to apply to a current state. In particular, as indicated in Lines 6-8, it prefers a service that produces the highest number of *new* parameters. In Line 9, if there is no applicable service that produces a higher number of new parameters, the application of such applicable services would not lead to a new state. Thus, search halts with no solution found. Otherwise, after the appropriate service is selected, a set of callable parameters and their associated semantic concepts are updated for a new state as shown in Line 11. The search continues until either a goal state is reached in that a given set of the desired output parameters of a composite service O can be obtained or we exhaust all service options and no solution for the semantic service composition problem can be found.

Procedure *Build-sequence*
Inputs: A web service repository *W,* a service ontology *KB,* a set of input (output) parameters I (O) of a composite service C
Output: An executable sequence of services *seq* that satisfies I and O

```
1  seq ← nil;
2  Current ← I ∪ infer-semantics(I) ; current state represents a set of
3  ;callable parameters and their related concepts derived from ontology
4  while W ≠ ∅ do
5      max ← 0
6      for each applicable service w to Current do; i.e., in(w) ⊆ Current
7          New_w ← out(w) – Current
8          if |New_w| > max then max ← |New_w|; service ← w
9      if max = 0 then return nil; no service can lead to a new state
10     Append service to seq
11     Current ← Current ∪ New_service ∪ infer-semantics(New_service)
12     if O ⊆ Current then return seq
13 ; goal state is reached since callable parameters in Current cover O
14                 else W ← W – {service}
15 return nil ;no sequence composition found
```

Figure 2. The *Build-sequence* algorithm.

We further refine the heuristic so that *Build-sequence* prefers a service that produces the highest number of (1) new desired output parameters (not shown here), and (2) new parameters when results from (1) are the same.

To improve performance of the algorithm, note that because of the monotone property of parameters and related concepts representing each state along the sequence of executable services, services that are applicable to a current state must also be applicable to the following states in the sequence. Thus, checking for the applicability of previously applicable services can be omitted and we can save time on Line 6 of Figure 2. Finally, for inference on the ontology, we implement a hash table to retrieve children of each concept in the ontology in a way that no table lookup is repeated. *Infer-semantics* recursively retrieves all successors of each concept. Since each table lookup takes a constant time for each concept, *infer-semantics* of a set of concepts in the ontology of m concepts takes $O(m)$ time.

B. Optimized Composition

The goal in this step is to eliminate unnecessary services in $seq = \langle w_1, w_2, ..., w_k \rangle$, which was obtained in Step (1), where $\delta(s_{i-1}, w_i) = s_i$ for $i = 1, \ldots, k$, $p(s_0) = I$ and $s_k \in G$, i.e., $O \subseteq p(s_k)$. It is clear that, for semantic composition, when web service w *produces* (*invokes*) parameter x, i.e., $x \in out(w)$, x and *infer-semantics*(x) are assumed to be callable parameters in the next state. Thus, we can compute $p(s_i) = p(s_{i-1}) \cup out(w_i) \cup infer\text{-}semantics(out(w_i))$ for $i = 1, \ldots, k$. We refer to any element of O as a *desirable* parameter.

Our pruning mechanism determines which web service in the *seq* is necessary for the requirements in a bottom up fashion. Suppose N is a collection of all necessary services found from the bottom of the path *seq* so far. Conceptually, a web service is *necessary* if it is the only service (among itself and all services in N) that produces a new parameter that is either (1) desirable or (2) a required input parameter of some service in N such that the required input parameter cannot be produced by any service in N.

To check if web service w_i that moves state s_{i-1} to state s_i on the *seq* path is necessary, we define (i) N_i, a set of all necessary services applied to *seq* after w_i, (ii) O_i, a set of desirable parameters produced by all services in N_i, and (iii) I_i, a set of input parameters required by all services in N_i such that they cannot be produced by any application of service in N_i in the *seq* order. Both O_i and I_i are used for testing the above condition (1) and (2), respectively, in order to determine if w_i is necessary or not. Figure 3 shows our proposed pruning conditions are checked in Lines 9 and 11, respectively.

Note that a set of desirable parameters produced by all service in N_i (necessary services w_j for $j > i$) is equivalent to a set of desirable parameters produced by all service in N_{i+1} and those that produced by w_{i+1} (and their inferred concepts), therefore we obtain Line 4 of Figure 3. Similarly, the necessary input parameters required by all services in N_i can be obtained by first finding the necessary input parameters re-

Procedure *Prune-sequence*
Inputs: O, a set of output parameters for a composite service;
$seq = \langle w_1, w_2, ..., w_k \rangle$ obtained from Build-sequence
Output: *seq* containing only *necessary* services (defined above).

```
1  for i ← k to 1
2      N_i ← {w_j | i < j ≤ k}; necessary services after applying w_i
3      E_i ← out(w_{i+1}) ∪ infer-semantics(out(w_{i+1}));
4      O_i ← O_{i+1} ∪ (O ∩ E_i); desirable parameters created by services in N_i
5      I_i ← (I_{i+1} – E_i) ∪ in(w_{i+1})
6              ; input parameters required by all services in N_i such that
7              ; none can be produced by (any service in) N_i in the seq order.
8      New ← p(s_i) – p(s_{i-1}); a set of new parameters produced by w_i
9      if New ∩ (O – O_i) ≠ ∅ ; there is a new parameter that
10                              ; is desirable but cannot be produced by N_i
11         or New ∩ I_i ≠ ∅ ; or that is a necessary input parameter for N_i
12     then w_i is necessary
13     else Eliminate w_i;
14 end for
```

Figure 3. The *Prune-sequence* algorithm.

quired by all services in N_{i+1} that cannot be produced by w_{i+1} (and their inferred concepts) and then adding $in(w_{i+1})$ as shown in Line 5. By making use of the results in a previous iteration, our approach provides an efficient pruning mechanism.

Although the pruning concept is not new, correct design of efficient pruning procedure can be subtle. For example, the required input parameters I_i defined as $\{x \in in(w) \mid w \in N_i\} - \{x \in out(w) \mid w \in N_i\}$ would be erroneous because it did not take the order of service applications into account.

C. Complexity Analysis

Let n be $|W|$, a number of web services in the repository, and m be a total number of concepts in the service ontology including parameters in the repository. To analyze the *Build-sequence* algorithm, recall that the callable parameters of states along any path in the state space search model exhibit a monotone property. Thus, each application of a new service to a state always yields a new state with a larger set of callable parameters (and corresponding inferred concepts). This monotone property implies that there is no backtrack required in the search since any applicable service to any previous state on the same path is also applicable to the next following states. Thus, the search can always move forward and *Build-sequence* guarantees to find a solution if it exists. Furthermore, the maximum number of states generated and traversed can be obtained from the longest possible (solution) path.

To determine the longest path generated by *Build-sequence*, we first note that the path is of finite length since P and W are finite and since a repetition of any service application will not lead to a new state. Therefore, any valid service composition sequence in our modeling approach contains no more than *all* services in the repository. This gives an upper bound of a valid service composite path length to be n. On the other hand, any path from an initial state to any state in a complete state space has length of at most m (see the longest path from the top to the bottom element of a complete lattice of m element set). From the above arguments, a valid sequence of our service composition problem has length at most $min(n, m)$. Since each state may be applicable to at most n services, the maximum search space is $n \cdot (min(n, m) + 1)$ and therefore, *Build-sequence* takes $O(n \cdot min(n, m) + m)$ time. The additional $O(m)$ time is contributed by *infer-semantics* as described in Section *A*. For the *Prune-sequence* algorithm, as shown in Figure 2, the for loop iterates at most $k \leq min(n, m)$ times and thus, the pruning step takes linear time, specifically $O(min(n, m))$. Thus, an overall semantic service composition has time complexity of $O(nm)$ for a repository of n services and a service ontology of m concepts.

V. An Illustration

Consider an online bookshop, where we want to develop a book order service application that lets a customer sign on a store account to find a purchase record including a tracking number and cost of a specific book. In particular the application has I/O specifications: I = {email, password, title, author} and O = {orderID, trackingID, cost}. Instead of developing this application from scratch, a developer decides to use web services available on the Internet.

I = {email, password, title, author}, O = {orderID, trackingID, cost}

Web service w	Input pars. $in(w)$	Out pars. $out(w)$
w_1: GetDiscountPrice	ISBN, status	discount cost
w_2: FindBook	title, author	ISBN
w_3: GetBookInfo	ISBN	title, author, year, publisher, cost
w_4: PurchaseBook	authenticated, authorized, ISBN	orderID
w_5: GetUserId	email	userID
w_6: Payment	credit card, price, shipping fee	authorized
w_7: Authenticate	userID, password	authenticated

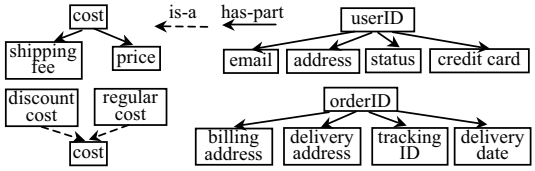
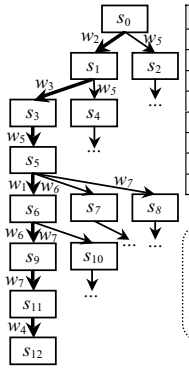


Figure 4. Service repository and service ontology.

Suppose a service repository of available services and service ontology are given at the top and bottom part of Figure 4, respectively. For example, the *Payment* web service, w_6, authorizes a specific book purchase if a customer's credit card can cover the charge. As indicated at the bottom left part of Figure 4, the ontology represents the fact that there are two types of cost (i.e., regular and discount), each of which combines the price of the book and its shipping fee. A hash table of keys (e.g., discount cost, regular cost) where each has associated concepts of its children is used to facilitate ontology inferences. Note that some concepts in the ontology do not appear in the repository (e.g., trackingID, billing address) and vice versa (e.g., authorized). In this scenario, since there is no single service in the repository that satisfies the composite service I/O requirements, we apply the *Build-sequence* algorithm to find appropriate executable combined services that meet the required specifications.



s	$p(s)$ & Inferred concepts	w: # new pars
s_0	{*e, pd, t, a*}∪∅	**w_2:1**, w_5:1
s_1	$p(s_0)$ ∪{*b*}∪∅	**w_3:4**, w_5:1
s_3	$p(s_1)$∪{*t, a, y, pb, c*}∪{*pr, sf*}	**w_5:1**
s_5	$p(s_3)$∪{*u-id*}∪{*e, ad, st, cc*}	**w_1:1**,**w_6:1**,w_7:1
s_6	$p(s_5)$∪{*dc*}∪∅	**w_6:1**, w_7:1
s_9	$p(s_6)$∪{*ao*}∪∅	w_7:1
s_{11}	$p(s_9)$∪{*ae*}∪∅	w_4:1
s_{12}	$p(s_{11})$∪{*o-id*}∪{*ba,da,t-id,dd*}	

a (author), *ad* (address), *ae* (authenticated), *ao* (authorized), *b* (ISBN), *ba* (billing address), *c* (cost), *cc* (credit card), *da* (delivery address), *dc* (discount cost), *dd* (delivery date), *e* (email), *o-id* (orderID), *pb* (publisher), *pd* (password), *pr* (price), *rc*(regular cost), *sf* (shipping fee), *st* (status), *t* (title), *t-id* (trackingID), *u-id* (userID), *y* (year)

Figure 5. Search valid composition sequence.

Figure 5 shows a partial search tree generated by *Build-sequence,* where corresponding callable parameters and inferred concepts of each state in the resulting sequence (using variables shown in a box at the bottom right corner) are given in a table on top right corner of Figure 5. For example, in the

first row of the table, the search starts from an initial state s_0 where its callable parameters are in $I = \{e, pd, t, a\}$ and no new additional concepts can be inferred from I from the given ontology. Since $in(w_2) = \{t, a\} \subseteq I$, w_2 is applicable to s_0 and so is w_5. These are the only applicable services to s_0. *Build-sequence* selects the first service found that produces new parameters the most. This is indicated by a service in boldface with a corresponding heuristic value after a colon. Here w_2 is selected with a new parameter b produced as shown in the second row of the table. The process continues until a state whose set of callable parameters covers O is found or no more services can be considered. In this scenario, a sequence $<w_2, w_3, w_5, w_1, w_6, w_7, w_4>$ is obtained. Next we apply the *Prune-sequence* algorithm to filter out unnecessary services.

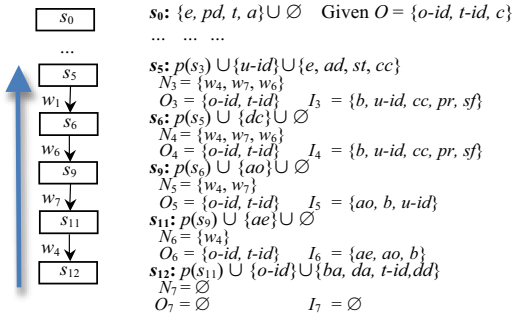


Figure 6. Pruning to minimize number of services to be deployed.

Figure 6 shows the results obtained by *Prune-sequence*. As shown in Figure 3, recall that for a service application level i, a service is necessary if its application results in a new parameter that is (1) desirable but cannot be produced by other following necessary services (i.e., in O but not O_i,) or (2) a necessary input parameter for other following necessary services (i.e., in I_i). Starting from the bottom service w_4, since N_7 is empty then $O_7 = \varnothing$. As shown in the second set (new parameters) and last set (new inferred concepts) after the label "s_{12}:" in Figure 6, since w_4 produces a new parameter *o-id* that can infer another new parameter *t-id*, both of which satisfies (1), thus w_4 is necessary.

Next we check if w_7 is necessary. Since w_7 produces a new parameter $ae \in I_6$, therefore (2) is satisfied and w_7 is necessary. Similarly, w_6 is necessary because there is at least one new parameter produced by w_6, namely $ao \in I_5$. To check if w_1 is necessary, the only new parameter produced by w_1 (see the second set after the label "s_6:") is *dc* and no other *new* parameters or concepts can be inferred (see the third set after the label "s_6:"). Since $dc \notin O - O_4$ *and* also $dc \notin I_4$, thus, w_1 does not satisfy both (1) and (2), and therefore it is not necessary. Similar process continues until we reach the first service application in the sequence. The resulting minimal sequence obtained is $< w_2, w_3, w_5, w_6, w_7, w_4>$.

VI. Experiments

Our analysis shows that the proposed approach is theoretically efficient. To further evaluate the approach, we conduct experiments on public benchmark data that are used for 2006 Web Service Challenge competition [10]. We compare our results with those obtained by Weise et al.'s approach [9], which won a championship of EEE06 (IEEE International Conference on e-Technology, e-Commerce and e-Service' 06) Web Service Challenge Competition [10]. Weise et al.'s system (will be referred to as WS) for optimizing semantic web composition is also available at [9].

We employ 10 service repository data sets, each of which contains (1) service ontology in XML Schema format, (2) a directory containing service descriptions, and (3) a test file consisting of one or more service composition problems. Each web service is described in WSDL and a composition request is expressed in XML. As shown in Table I, each of the Repositories 1-6 contains a single composition problem, whereas Repository 7, 8, 9, and 11 contains four, eight, 10 and three problems, respectively. This gives a total of 31 composition test problems. We ran experiments with both approaches on a PC with Core 2 Duo (1.67GHz), Window Vista Home Premium and 3 GBs RAM.

Table I shows our experimental results, where the last two columns (columns 7 and 8) compare the running times between our approach and WS. The better running times are in boldface. The average result of each repository data set is summarized in a shaded row below results of each of the composition problems in the repository. As noted earlier, each

TABLE I. COMPARISONS WITH EEE06 CHAMPION.

			Ours				WS
Test	**#Serv.**	**#Params.**	**Before**	**After**	**%reduct.**	**Time(ms)**	**Time(ms)**
1	1000	56210	5	5	0%	**119**	168
2	1000	56210	18	12	33%	**1454**	3100
3	10000	58254	16	10	38%	**14224**	55702
4	2000	58254	17	15	12%	**3511**	8546
5	4000	58254	35	30	14%	27668	**22857**
6	8000	58254	50	40	20%	81496	**45432**
7.1	118	1590	3	2	33%	**3**	22
7.2	118	1590	2	2	0%	**2**	90
7.3	118	1590	7	3	57%	**5**	14
7.4	118	1590	4	4	0%	**6**	13
7	118	1590	4	2.8	23%	**4**	34.8
8.1	480	15540	5	2	60%	**19**	21
8.2	480	15540	2	2	0%	**7**	22
8.3	480	15540	5	3	40%	17	15
8.4	480	15540	8	4	50%	**36**	14
8.5	480	15540	3	3	0%	**14**	22
8.6	480	15540	2	2	0%	**9**	14
8.8	480	15540	2	2	0%	**6**	18
8.9	480	15540	3	3	0%	**10**	19
8	480	15540	3.8	2.6	19%	**14.8**	18.1
9.1	978	979740	3	2	33%	**15**	23181
9.3	978	979740	6	3	50%	**47**	18998
9.4	978	979740	4	4	0%	**34**	22936
9.5	978	979740	3	3	0%	**20**	21444
9.6	978	979740	2	2	0%	**13**	16527
9.7	978	979740	4	4	0%	**60**	15596
9.9	978	979740	3	3	0%	**22**	16311
9.10	978	979740	4	4	0%	**45**	15626
9.11	978	979740	3	3	0%	**25**	16409
9.12	978	979740	6	4	33%	**64**	16234
9	978	979740	3.8	3.2	12%	**34.4**	18,326.2
11.1	4000	10890	11	8	27%	**1161**	3433
11.3	4000	10890	8	6	25%	**548**	2934
11.5	4000	10890	5	4	20%	**154**	2979
11	4000	10890	8.0	6.0	24%	**621.2**	3115.3
Avg	1680.4	332,457.6	8.0	6.3	18%	4,219.8	10,603

of the first six repository data sets contains a single composition problem and therefore there is no result for other composition problems shown for these data sets. Columns 4 and 5 indicate the number of services in the service composition solution obtained by *Build-sequence* and *Prune-sequence,* respectively. Column 6 shows a percentage of reduction in the number of services in the composition as a result of our minimization by pruning.

On the average, our approach reduces the number of unnecessary services by 23%, 19%, 12%, 24% in Repository 7, 8, 9 and 11, respectively. In general, our approach reduces the number of unnecessary services by 18% on the overall average over 31 composition problems. On correctness, our approach produces a composition solution with correct number of minimal services for each problem in every repository data (not shown here). Thus, its correctness is 100%.

On the running times, as indicated in the last row of Table I, our approach significantly reduces the average running time, compared to WS, by 60.2%. Our approach performs better than WS 29 out of 31 test cases. On the other hand, for the Repositories 5 and 6, WS only reduces the running times, compared to ours, by an average of 37.4%. Since the most time consuming step for our approach is finding applicable services to a current state during the search for service composition solution in *Build-sequence,* our approach tends to perform poorly when a solution depth is long. As shown in Table I, composition solutions in Repositories 5 and 6 require at least 30 and 40 services, which are significantly more than other cases. Furthermore, most other approaches including WS have implemented sophisticated frameworks and mechanisms (e.g., indexing, caching) to improve performance efficiency.

The most striking results are those obtained in Repository 9, where our approach has dramatically gained speed up in the running times in each composition problem by three orders of magnitude. In this data set, our approach reduces the average running time, compared to WS, by 99.8%. One characteristic of this repository is an extremely large number of parameters in the repository. We conjecture that the number of parameters (including concepts in the ontology) does not negatively impact the running time as much as the number of services. This is to be expected for semantic web service composition. The results obtained in this data set support the strength of our approach when semantic inferences are demanding. Considering the fact that WS is a champion of EEE06 Web service challenge competition, our approach performs very well.

VII. CONCLUSION

This paper presents a simple modeling approach together with two efficient heuristic algorithms for developing a composite web service application from a given service repository and service ontology. The approach focuses on using knowledge in the ontology to help construct the composition structure. It automatically searches for appropriate combination of (near) minimum number of executable services to satisfy the input and output requirements of the application to be developed. We describe complexity analysis of our approach and experiments to evaluate the approach in practice. Our empirical results on the majority of public benchmark data sets outperform those obtained by a champion of EEE06 Web service challenge competition despite the fact that no sophisticated efficiency mechanism has been exploited.

Current inference mechanism for semantic composition is simple in that it deals with inheritance and specification of part-whole relationships. In general, it is desirable to enable aggregation of part-whole relationships. This would increase possible options for valid construction that can enhance composition of semantic web services. Future work includes (1) enhancement of this inference capability, (2) improvement of some efficiency mechanisms as used in other systems, and (3) incorporation of quality of service factors including non-functional features such as security into the selection of appropriate web services during the service composition.

REFERENCES

[1] M. Beek, A. Bucchiarone and S. Gnesi, "Web Service Composition Approaches: From Industrial Standards to Formal Methods", in *Proc. of Conf. on Internet and Web App. and Services* (ICIW'07), IEEE Com. Soc. Press, Mauritius, 2007.

[2] J. Hoffmann, J. Scicluna, T. Kaczmarek, I. Weber, "Polynomial-Time Reasoning for Semantic Web Service Composition," *in Procs. of IEEE Inter. Conf. on Service*s, pp.229-236, 2007

[3] S. McIlraith, T. Son, "Adapting Golog for Composition of Semantic Web Services," *in Procs. of IEEE Inter. Conf. on Knowledge Representation and Reasoning KR'02*, pp. 482-493, 2002.

[4] N. Milanovic, and M. Malek, "Current solutions for Web Service Composition", *Internet Computing*, IEEE Computer Society Press, 8(6): 51-59, 2004.

[5] S. Oh, J. Yoo, H. Kil, D. Lee, and S. Kumara, "Semantic Web Sevice Discovery and Composition Using Flexible Parameter Matching" *in Proc. of IEEE Joint Conf. CEC/EEE,* pp. 533-542, 2007.

[6] J. Rao and X. Su, "A Survey of Automated Web Service Composition Methods", in *Workshop on Semantic Web Services and Web Process Composition,* Springer-Verlag, pp. 43-54, 2004.

[7] B. Srivastava and J. Koehler, "Web Service Composition - Current Solutions and Open Problems", *ICAPS 2003 Workshop on Planning for Web Services,* 2003.

[8] S. G. H. Tabatabaei, W. M. N. W. Kadir, S. Ibrahim, "Semantic Web Service Discovery and Composition Based on AI Planning and Web Service Modeling Ontology," *in Procs. of IEEE Inter. Conf. on Services Computing APSCC'08*, pp.397-403, 2008.

[9] T. Weise, S. Bleul, D. Comes, K. Geihs, "Different Approaches to Semantic Web Service Composition," in *Procs. of IEEE Inter. Conf. on Internet and Web App. and Services*, pp. 90-96, 2008 (software: http://www.it-weise.de/documents/files/BWG2007WSC_software._zip, available 2009).

[10] WS-Challenge 06, http://ws-challenge.georgetown.edu/ws-challenge/wsc06/, available 2009.

[11] R. Zhang, I. Arpinar and B. Aleman-Meza, "Automatic Composition of Semantic Web Services," in *Proc. of Inter. Conf. on Web Services,* ICWS '03, Las Vegas, Nevada, pp. 38-41, 2003.

[12] Y. Zhang, T. Yu, K. Raman, and K. Lin, "Strategies for efficient syntactical and semantic web service discovery and composition", *in Proc. of IEEE Joint Conf. CEC/EEE,* pp. 452-454, 2007.